CHAPTER

THIRTEEN

CONCURRENT PASCAL AND DISTRIBUTED PROCESSES

One of the most prolific researchers and implementors of concurrent and distributed systems is Per Brinch Hansen. This chapter describes Brinch Hansen's language for distributed programming, Distributed Processes. For pedagogical purposes, we begin by describing a predecessor of Distributed Processes, Concurrent Pascal.

Concurrent Pascal is a multiprocessing extension of Pascal. It has three important features for structuring concurrency: processes, monitors, and classes. Processes are active computing agents. Monitors synchronize access to shared data. Classes provide structured access to data when synchronization is not required.

Distributed Processes is a language for distributed and real-time systems. It takes the processes, monitors, and classes of Concurrent Pascal and unifies them into a single construct, the process. Its communication mechanism, the remote procedure call, has been copied by several other systems.

These languages focus on the problems of resource management and real-time control. Their "pragmatic" features are the pragmatic features of the system implementor—synchronization and primitive abstraction mechanisms—but not dynamic structures, automatic buffering, and so forth. The primary motivation in the design of Concurrent Pascal and Distributed Processes is to facilitate the programming of powerful, secure, and easily extended systems programs.

13-1 CONCURRENT PASCAL

Concurrent Pascal extends the sequential programming language Pascal with mechanisms for structured multiprocessing. Its design was motivated by the observation that the hardest part of concurrent programming is assuring the security of local storage and mutual exclusion in accessing shared storage. Thus, in Concurrent Pascal access rights and synchronization are primitive language structures, enforced by the compiler.

Concurrent Pascal has three structures that combine aspects of active computing and static data storage: processes, monitors, and classes. A *process* is a computing agent. It has three parts: a sequential program, private data, and access rights. Concurrent Pascal rejects the scoping rules of languages like Algol. A process's private data is the only data it can access directly; no other process can access that data. The access rights of a process specify the other system objects (monitors and classes) that this process can call. A *monitor* is a protection and abstraction mechanism for shared data. Monitors ensure that only a single process acts on shared data at any time. *Classes*, like monitors, provide data abstraction. But unlike monitors, the structure of programs guarantees that only a single process will execute the code of any class at any time.

Our first example in Concurrent Pascal is a program for printing the prime Fibonacci numbers. This program has two processes, one that generates successive Fibonacci numbers, and another that tests them for primeness. These processes communicate through a shared buffer—a monitor. This buffer stores a single number. When the prime tester finds that one of the Fibonacci numbers sent it is prime, it calls on a class object, a LinePrinter, to print it. Figure 13-1 shows the parts of this program. We describe each kind of object in a **type** statement, declaring actual instances of each type that the program uses in a **var** statement. The program of the Fibonacci process is as follows:

```
type Fibonacci =
process (buf: buffer)                  -- A Fibonacci process has access rights to a
                                          monitor of type buffer.
var this, last, previous: integer;     -- the process's private data
begin
    last := 0;
    this := 1;
    cycle                              -- that is, "while true do"
        previous := last;              -- Compute the next Fibonacci number.
        last     := this;
        this     := last + previous;
        buf.add(this)                  -- Place this Fibonacci number in the buffer.
    end
end;
```

Figure 13-1 Information flow in communication in the prime-Fibonacci system.

Processes do not share data directly. Instead, they share data through calls on monitors [Hoare 74]. A monitor specifies a shared data structure and provides the procedures and functions to manipulate that data structure. A monitor procedure that can be called from another process or monitor is a *procedure entry*. For example, monitor stack could have procedure entry pop. Processes could pop an element from the stack by calling stack.pop.

Each monitor also has an initial operation. When it creates a new monitor, the system executes that monitor's initial operation. An important use of this operation is to initialize the monitor's data structures. For example, the initial operation of a stack monitor would set the stack to empty. Like a process, a monitor can also have explicit access rights to other monitors and classes.

To enter a monitor, a process calls an entry procedure in that monitor. If no other process is in that monitor, the process begins executing the code associated with that entry. If another process is in the monitor, the calling process waits until the monitor is free. The restriction that only a single process can execute the code of a monitor at any time is a simple mechanism for short-term scheduling.

Monitors serialize access to shared data. However, many programs require a more complex scheduling algorithm than simple mutual exclusion. Concurrent Pascal provides queues for medium-term scheduling. Queues are used to delay processes until it is appropriate for them to continue. There are two operations on queues, delay and continue. A queue stores a single process-state descriptor. If process A executes **delay** on queue q, then A is blocked on q. The monitor unlocks and allows other processes entry. When another process, B, executes **continue**(q), B returns from its call to the monitor and A continues executing (in the monitor) from the point after the **delay** statement. The monitor remains locked against other processes. Despite the mental image of processes waiting in line that the name "queue" evokes, queues in Concurrent Pascal can hold only a single process. However, Concurrent Pascal allows arrays of queues.

In our example, the buffer is a monitor. Buffers do not have access rights to any other system objects. They have storage for a single buffered value, an

integer; a boolean flag that indicates that the buffer is full; and two queues, one which delays a consumer that tries to remove from an empty buffer and another which delays a producer that tries to add to a full buffer. Buffers have two entry procedures: add, called by the producer, and remove, called by the consumer.*

```
type buffer =
monitor;
var
    data                : integer;
    flag                : boolean;
    pwaiting, cwaiting  : queue;

procedure entry add (invalue: integer);
begin
    if flag then delay(pwaiting);          -- If the buffer is full, delay the
    flag  := true;                            producer.
    data  := invalue;
    continue(cwaiting)
end;

procedure entry remove (var outvalue: integer);
begin
    if not(flag) then delay(cwaiting);     -- If the buffer is empty, delay the
    flag     := false;                        consumer.
    outvalue := data;
    continue(pwaiting)
end;

begin                                      -- initial statement
    flag := false                          -- The buffer is initially empty.
end;
```

The consumer process PrimeTester resembles the producer. It calls the buffer to obtain the next value and tests to see if it is prime. If it is, the consumer process calls the printer, a class object of type LinePrinter, to print it.

```
type PrimeTester =
process (buf: buffer);
var
    num, j : integer;
```

* Variables declared globally in a monitor are permanent and shared by the entry procedures of that monitor. Variables declared in a procedure are allocated afresh for each call to that procedure. Monitors can also have "non-entry" (ordinary) procedures, used by the program of that monitor and not by other system objects.

```
    prime  : boolean;
    printer : LinePrinter;    -- This declaration creates a printer class object, of
                                 type LinePrinter.
begin
    cycle
        buf.remove(num);
        if ((num mod 2) = 1 or (num = 2)) then    -- odd or 2
        begin
            j      := 3;
            prime := true;
            while (j < sqrt(num)) and prime do    -- not the most efficient
                prime := not ((num mod j) = 0);      way to test primeness
                j      := j + 2
            end
        end;
        if prime then printer.show(num)
    end
end;
```

In Concurrent Pascal, a *class* is an abstract data object that is not shared. A class object can be declared only as a permanent variable within another system object. Classes can be passed as (access-rights) parameters to other classes, but never to processes or monitors. Hence, two processes cannot call the same class object simultaneously and class objects do not require scheduling. This permits the Concurrent Pascal compiler to optimize calls on classes, making such calls execute faster than calls on monitors. This optimization is the major reason for including classes in Concurrent Pascal. The difference between classes and monitors is primarily one of efficiency, not functionality.

Peripheral devices are treated as hardware implementations of monitors. They have only a single access procedure, **io**. This procedure delays the calling process until the completion of the input-output process. Thus, a class of type LinePrinter is

```
type LinePrinter =
class;
var parm = record ... end;
  -- The standard procedure io takes arguments of a particular internal
     structure. We omit the details of that structure.

procedure entry show (i: integer);
begin
    io(i, parm, "LPT")
end;
```

```
begin
  -- LinePrinters do not need initialization.
end;
```

Each of the above examples is a type declaration. Specifying a description of a monitor or a process does not create one, any more than specifying a type declaration in Pascal allocates storage. A program with such type statements allocates these objects in a **var** statement. The program starts the processes and runs the initialization statements of the monitors and classes with an **init** statement. This **init** statement also provides the names of the other system objects to which this object has access. The entire program is

```
program FibonacciPrimes;
type
    Fibonacci   = process ... ;
    buffer      = monitor ... ;
    PrimeTester = process ... ;
var
    FProd   : Fibonacci;
    FConsum : PrimeTester;
    FBuffer : buffer;
begin
    init FProd(FBuffer), FConsum(FBuffer), FBuffer;
end;
```

Storage allocation in Concurrent Pascal is completely static. It lacks recursion and has no command to dynamically create new processes or monitors. Concurrent Pascal not only does not dynamically allocate storage, it never deallocates storage. Even if a process has terminated, its storage continues to exist. The system cannot reclaim its storage because that storage may have been passed by reference to another system object.

Must Concurrent Pascal be so static? Brinch Hansen presents the reasoning behind these choices as [Brinch Hansen 75, p. 201]:

> Dynamic process deletion will certainly complicate the semantics and implementation of a programming language considerably. And since it appears to be unnecessary for a large class of real-time applications, it seems wise to exclude it altogether. So an operating system written in Concurrent Pascal will consist of a fixed set of processes, monitors and classes. These components and their data structures will exist forever after system initialization. An operating system can, however, be extended by recompilation. It remains to be seen whether this restriction will simplify or complicate operating system design.

Dining philosophers The dining philosophers problem illustrates processes that share monitors. Our program uses five processes, one for each philosopher; a monitor for each fork; and a monitor for the room. A philosopher process thinks, enters the room, picks up the forks, eats, drops the forks, leaves the room, and

Figure 13-2 The objects in the dining philosophers program.

repeats the cycle. The dining philosophers problem is a pure synchronization problem. Thus, the system components exchange only synchronization, not information. Figure 13-2 shows the communication relationships of the elements of this system.

```
type philosopher =
process (theroom: room; left, right: fork);
begin
    cycle
        -- think;
        theroom.enter;
        left.pickup;
        right.pickup;
        -- eat;
        left.putdown;
        right.putdown;
        theroom.exit
    end
end;
```

Forks are monitors. Each fork has a boolean flag that shows if it is taken, a queue to delay the philosopher that tries to take it when it is busy, and two entry procedures, pickup and putdown. Forks are initially free. A philosopher that tries to pick up a taken fork is delayed in the pleasewait queue; each philosopher, as she drops the fork, continues that queue, thereby giving a waiting philosopher her turn.

```
type fork =
monitor;
var
    taken      : boolean;
    pleasewait : queue;

procedure entry pickup;
begin
    if taken then delay(pleasewait);
    taken := true
end;

procedure entry putdown;
begin
    taken := false;
    continue(pleasewait)
end;

begin
    taken := false
end;
```

A room is a monitor. It keeps the number of dining philosophers at four or fewer, delaying any philosopher that tries to enter when four of her companions are eating. It has a variable to count the philosophers in the room, a queue for the waiting philosopher, and procedure entries enter and exit. The room is initially empty.

```
type room =
monitor;
var
    occupancy: integer;
    WithoutReservations: queue;

procedure entry enter;
begin
    if occupancy = 4 then delay (WithoutReservations);
    occupancy := occupancy + 1
end;
```

```
procedure entry exit;
begin
    occupancy := occupancy − 1;
    continue (WithoutReservations)
end;

begin
    occupancy := 0
end;
```

Since processes, monitors, and classes are objects, declared in type statements, we can have arrays of them. We pass the objects they access to the philosophers when we initialize them. The program for the dining philosophers problem is

```
program dining;
type
    philosopher = process ... ;
    fork        = monitor ... ;
    room        = monitor ... ;
var
    philosophers : array [0 .. 4] of philosopher;
    forks        : array [0 .. 4] of fork;
    chamber      : room;
    i            : integer;

begin
    init chamber;
    for i := 0 to 4 do init forks[i];
    for i := 0 to 4 do init philosophers[i](chamber, forks[i], forks[(i+1) mod 5])
end;
```

13-2 DISTRIBUTED PROCESSES

Concurrent Pascal provides three different primitives (processes, monitors, and classes) for data encapsulation and parallel processing. Brinch Hansen recognized that this multiplicity was unnecessary. The difference between monitors and classes is primarily an optimization hint to the compiler. And the difference between processes and monitors is just the embedding of active processing in processes. Even so, monitors and classes need some active processing for their initialization statements. In his successor language, Distributed Processes, Brinch Hansen unifies these three concepts into a single entity, the process.

Concern for resource allocation and real-time issues motivated the design of Distributed Processes. A Distributed Processes system has a fixed set of concurrently executing, sequential processes. Processes are determined at compilation and can be neither dynamically created nor destroyed.

A Distributed Processes's process can access only its own local storage. There are no global data structures (like the monitors of Concurrent Pascal) shared by several processes. Instead, processes communicate by calling procedures (*common procedures*) in other processes, sending and returning parameter values. Each process multiprocesses the tasks of executing its own program and handling calls to its common procedures. In some sense, the processes of Distributed Processes act as monitors for each other, though without the specific synchronization rules of monitors. Since Distributed Processes is concerned with distributed processing, values are passed by value, not by reference. A call from one process to another is an *external request*.

Distributed Processes uses Pascal for syntactic foundation. The principal extensions are the constructs for interprocess communication. Each process has four parts: a name, local storage, common procedures, and an initial statement. Syntactically, the verb **call** invokes an external request. Like Concurrent Pascal, a process calls a procedure in another process by referencing the procedure name together with the process name. Thus, the one parameter procedure NextCharacter in process CardReader is invoked by

call CardReader.NextCharacter(C)

Procedure NextCharacter in process CardReader has no input (value) parameters and a single output (result) parameter of type char. The process's input and output parameters are separated in the parameter list declaration by a #. Thus, the declaration of process CardReader begins

```
process CardReader;
    var count: integer;                      -- a local variable
    procedure NextCharacter (# ch: char);    -- a single output parameter
                ⋮
```

Distributed Processes includes a variant of Dijkstra's guarded commands (Section 2-2). *Guarded clauses* are formed by joining the guarded condition (a boolean expression) to the guarded action (a statement) with a ":". Guarded clauses are joined with "|"s to form *guarded regions*.* Of course, guarded regions imply an indeterminate choice among the open guarded clauses.

Each process performs two kinds of computations: executing its own program (its initial statement) and handling calls to its common procedures. The

* This contrasts with the $\rightarrow$ and $\square$ notation of the original syntax.

Table 13-1 Guards and loops

	Non-waiting	Waiting
Single Execution	**if** $B_1:S_1 \mid \ldots \mid B_n:S_n$ **end** If a B_i is true, then execute the corresponding S_i; an error if none of the B_i is true.	**when** $B_1:S_1 \mid \ldots \mid B_n:S_n$ **end** Wait for a B_i to be true, then execute the corresponding S_i.
Repeated Execution	**do** $B_1:S_1 \mid \ldots \mid B_n:S_n$ **end** Repeatedly find a true B_i and execute the corresponding S_i, until all B_i are false.	**cycle** $B_1:S_1 \mid \ldots \mid B_n:S_n$ **end** Repeatedly find a true B_i and execute the corresponding S_i. If no B_i is true, wait until one is. This statement never terminates.

process interleaves these actions. This interleaving is not preemptive; instead, the process executes each task until the task blocks in a guarded command. At that point, the process can execute another task. Specifically, the process begins by executing its initial statement. When this statement terminates or blocks, the process starts some other pending operation. When that operation terminates or blocks, the process starts yet another pending operation. These operations are either resumptions of the initial statement or calls to the process's procedures. Operations blocked in guarded commands become pending when one of their guards becomes true. This interleaving continues for the life of the program. Even if the initial statement terminates, the process continues to exist, handling calls to its common procedures. Distributed Processes does not guarantee any particular ordering on the interleaved operations of a process. We know only that the first statement executed is the process's initialization statement. The interleaving is not preemptive. It is a function of the execution path of the program, not the pseudosimultaneity of simulated multiprocessing.

In Distributed Processes, guarded commands control two dimensions of processing: waiting and repetition. Distributed Processes has two choices for each of these and a language verb for each of the four possible combinations. Waiting concerns the action to be taken when none of the guard clauses is true. In that case, the process can either wait for one to become true (by using the language verbs **when** and **cycle**) or exit the statement (**if**, **do**). Repetition specifies how frequently to evaluate the guarded region: once (**if**, **when**) or repeatedly (**do**, **cycle**). The **do** statement executes until all guards are false; the **if** statement aborts the program with an error if all guards are false. Table 13-1 summarizes the kinds of guarded statements in Distributed Processes.

A process executes the statements of its current program segment until either (1) the program blocks in the guarded region of a **when** or **cycle** statement, or (2) the program blocks, waiting for the return from a call to an external procedure. If the process is in a guarded region, then it is free to interleave the

evaluation of the initial statement and other calls to its common procedures. On the other hand, if the process is waiting on an external call, the process pauses until that call returns. That is, a process blocked on a guarded command is waiting to serve and is eligible to handle other calls. A process blocked on a call to another process is presumed to need the results of that call before it can continue. When the external call returns, the process continues executing statements where it left off. This implies that processes must not be mutually recursive; if process A calls a procedure in process B and process B then calls a procedure in process A, they are both blocked, each waiting for the other's return.

Unlike CSP (Chapter 10), guarded commands in Distributed Processes do not specifically control communication. Instead, a process pauses in a guarded command, waiting for changes caused by other calls to this process.

Binary semaphore Perhaps the simplest synchronization primitive is the binary semaphore. In our Distributed Processes program for a semaphore, the semaphore is a process. It has two common procedures, P (get the semaphore) and V (release the semaphore). The semaphore keeps its state in variable s. When s is positive, the semaphore is free; when it is zero, the semaphore is busy. Procedure P waits until s is positive, then decrements it and continues. Procedure V simply increments s.

```
process Binary_Semaphore;
var s: integer;

procedure P;
    when s > 0:
        s := s − 1
    end;

procedure V;
    s := s + 1;

begin    -- initialization statement
    s := 1
end;
```

A general semaphore that permits n processes to share a resource is the binary semaphore with s initialized to n.

Dining philosophers In Distributed Processes, we can declare an array of processes, all executing the same program but each with its own storage. Identifier **this**, when used in the body of a process, is the index of that process in the array. Our program for the dining philosophers problem uses an array of five philosopher processes, five fork processes, and a room process. Figure 13-3 shows the calling relationships in the program. The program for a philosopher is as follows:

Figure 13-3 The Distributed Processes dining philosophers.

```
process philosopher [5];     -- There are five philosophers.
  -- Philosophers have no storage. Since they are not called by other
     processes, they do not have entry procedures.

do true:                     -- one way to get an infinite loop
     -- think;
    call room.enter;
    call fork[this].pickup;
    call fork[(this + 1) mod 5].pickup;
     -- eat;
    call fork[this].putdown;
    call fork[(this + 1) mod 5].putdown;
    call room.exit;
end;
```

Forks are also processes. They keep track of their state in boolean variable busy.

```
process fork[5];
```

```
var busy: boolean;

procedure entry pickup;
    when not(busy):
        busy := true
end;

procedure entry putdown;
    busy := false

  -- initialization statement
    busy := false
end;
```

The room keeps the usual counts.

```
process room;
var occupancy: integer;

procedure entry enter;
    when occupancy < 5:
        occupancy := occupancy + 1
    end;

procedure entry exit;
    occupancy := occupancy − 1
end;

  -- initialization statement
    occupancy := 0
end;
```

The last section showed a similar solution to the dining philosophers problem in Concurrent Pascal. That solution relied on explicit **delay** and **continue** statements to schedule the philosopher processes. On the other hand, this solution uses the indeterminacy of guarded commands for scheduling.

Apart from the extensions described above, the syntax of Distributed Processes is just a variant of standard Pascal. However, like Concurrent Pascal, Distributed Processes does not have any constructs (like recursion and explicit allocation) that dynamically create storage. Therefore, storage allocation in Distributed Processes can be done at compilation.

Distributed Processes is a language for implementing resource managers. It requires that the conceptual processes of the programming language must be matched, one for one, with the physical processors of the distributed system.

Bounded buffer A bounded buffer (that stores elements of type BufferItems) is a process with two procedures, Insert and Remove. Insert waits until the buffer is not full; Remove, until the buffer is not empty. They interact by updating pointers into the buffer.

```
process BoundedBuffer;
const Bufsize = 100;
var
    first, last : integer;    -- First points to the next available item; last to the
                                 most recent addition.
    queue       : array [0 .. Bufsize − 1 ] of BufferItem;

procedure Insert (m: BufferItem);                    -- one input parameter
    when not (((last + 1) mod Bufsize) = first):     -- queue not full
        last          := (last + 1) mod Bufsize;     -- Put this item in the
        queue[last]   := m                              queue.
    end;

procedure Remove (# m: BufferItem);      -- one output parameter
    when not (last = first):             -- queue not empty
        m     := queue[first];           -- Pull an item from the queue.
        first := (first + 1) mod Bufsize
    end;

begin                                    -- initialization statement
    first := 0;
    last  := 0
end;
```

The system executes the initialization statement of the buffer once. On the other hand, the buffer exists for the entire run of the program.

This is an antifair buffer. A process that wishes to access this buffer can be arbitrarily and indefinitely ignored while the buffer handles other requests.

Perspective

Concurrent Pascal and Distributed Processes lie on the extreme operating systems end of the coordinated computing spectrum. These languages have an imperative, statement-oriented syntax, primitives that implement mutual exclusion, explicit processes that cannot be dynamically created, and predefined connections between processes. They rely on strong typing and other compilation checks to ensure program correctness. Brinch Hansen views such checks as the crucial ingredients for developing efficient concurrent computing systems. He writes [Brinch Hansen 78, p. 934]:

> Real-time programs must achieve the ultimate in simplicity, reliability, and efficiency. Otherwise one can neither understand them, depend on them, nor expect them to keep pace with their environments. To make real-time programs manageable it is essential to write them in an abstract programming language that hides irrelevant machine detail and makes extensive compilation checks possible. To make real-time programs efficient at the same time will probably require the design of computer architectures tailored to abstract languages (or even to particular applications).

The evolution of these languages (from Concurrent Pascal through Distributed Processes and on to Edison, discussed in the bibliography) moves away from language design based on the perceived requirements of compiler construction and towards building generality into the language. Concurrent Pascal explicitly distinguishes between active processing elements and passive shared structures and between synchronized and unsynchronized structures. Distributed Processes eliminates this distinction. It has only a single variety of object, the process. The language that results turns out to be not only simpler and more esthetically pleasing, but also a system for which it is easier to write a compiler. Brinch Hansen states [Brinch Hansen 78, p. 940]:

> The Concurrent Pascal machine distinguishes between 15 virtual instructions for classes, monitors, and processes. This number would be reduced by a factor of three for Distributed Processes. In addition, numerous special cases would disappear in the compiler.

By and large, these languages are designed to be practical, usable tools, instead of simple academic exercises. Many of the decisions in their design and implementation were based on the difficulty of system implementation or requirements for explicit user control. These decisions have resulted in theoretical flaws—the lack of recursion, the static storage allocation, and the fixed process structure being among the most critical. However, it is inappropriate to urge theoretical nicety on someone who must get something to work. The computing world is littered with impractical implementations of ideas that are esthetically pleasing. (Of course, the computing world is also littered with impractical systems that ignored theoretical generality chasing after the chimera of efficiency.)

PROBLEMS

13-1 Rewrite the Concurrent Pascal buffer program to use a larger buffer.

13-2 Rewrite the Concurrent Pascal bounded buffer program to serve more than one producer and more than one consumer.

13-3 Redesign the bounded buffer programs in Concurrent Pascal and Distributed Processes so that a producer and a consumer can concurrently update the buffer. How many monitors or processes does your solution use?

13-4 Can a philosopher starve in the Concurrent Pascal solution to the dining philosophers problem?

13-5 Can a philosopher starve in the Distributed Processes solution to the dining philosophers problem?

13-6 Program a manager for the readers-writers problem in Distributed Processes.

REFERENCES

[**Brinch Hansen 75**] Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Trans. Softw. Eng.*, vol. SE-1, no. 2 (June 1975), pp. 199–207. This paper is a brief description of the language Concurrent Pascal. Brinch Hansen illustrates the language with examples of the buffer processes of a miniature operating system.

[**Brinch Hansen 77**] Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, New Jersey (1977). Brinch Hansen describes the nature of synchronization and the languages Pascal and Concurrent Pascal. He then gives several examples of concurrent systems written in Concurrent Pascal.

[**Brinch Hansen 78**] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *CACM*, vol. 21, no. 11 (November 1978), pp. 934–941. This paper describes Distributed Processes. The binary semaphore and dining philosophers programs are derived from this article.

[**Brinch Hansen 81**] Brinch Hansen, P., "The Design of Edison," *Softw. Pract. Exper.*, vol. 11, no. 4 (April 1981), pp. 363–396. The path from Concurrent Pascal to Distributed Processes was marked by reduction and simplification—principally, the unification of the monitors, classes, and processes of Concurrent Pascal into a single, distributable object, the process. In Edison, Brinch Hansen takes this process one step further, omitting most conventional programming statements and synchronization structures.

Edison transforms the processes of Distributed Processes into Modules. Modules can allocate storage and declare procedures and other modules. Each module has an initial operation that is executed when the module is created. However, modules do not enforce mutual exclusion. Several processes can be executing the procedures of the same module simultaneously. Modules achieve mutual exclusion by using conditional critical regions. Only one module can execute in the "global" conditional critical region at any time. Concurrency is indicated with the equivalent of a **parbegin** statement.

Edison attempts to provide the tools for constructing concurrent systems, not to dictate the tools that must be used. The processes of Distributed Processes combine mutual exclusion and data abstraction. Edison separates these notions into explicit mutual exclusion (conditional critical regions) and data abstraction (modules). Applications that require monitors can implement them using modules and conditional critical regions.

Edison makes several linguistic advances over its predecessors. In particular, Edison permits procedures as procedure parameters and allows recursive procedure calls. (This second feature requires Edison to do dynamic storage allocation.) In addition, Brinch Hansen proposes an interesting addition to the syntax of typed languages, retyped variables. If `x` is a variable and `t` a type, the expression `x:t` is the value of the bit string that is `x` in the type `t`. The storage size of objects of the type of `x` and of objects of type `t` must be the same.

This issue of *Software—Practice and Experience* contains papers by Brinch Hansen describing Edison and giving examples of Edison programs.

[**Hoare 74**] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *CACM*, vol. 17, no. 10 (October 1974), pp. 549–557. This paper is Hoare's original description of monitors. He argues that monitors are useful in programming operating systems.

[**Li 81**] Li, C.-M., and M. T. Liu, "Dislang: A Distributed Programming Language/System," *Proc. 2d Int. Conf. Distrib. Comput. Syst.*, Paris (April 1981), pp. 162–172. Li and Liu propose the language Communicating Distributed Processes (CDP). CDP extends Distributed Processes to be more "distributed." More specifically, CDP supplements Distributed Processes with the following additions: (1) A process can specify an action to be taken on communication time-out. This action can be to retry the communication, to abort the communication, or to transfer control to an exception routine. (2) A process can use one of several different broadcast mechanisms to communicate with several processes

in the same step. (3) The language supports both synchronous and asynchronous requests. (4) A program can specify that an operation is "atomic." Failure in an atomic action returns the system to its state before the action was begun. (See Section 17-2 for a language based on atomic actions.) (5) Timestamps are a primitive system data type. The system generates new timestamps on request. (6) The broadcast mechanism allows the creation of several responses to a single request. Programs can specify which of these responses are desired: the first, the last, or all of them. And (7) the system can automatically create replicated copies of data (for replicated databases).

To illustrate the features of the language, Li and Liu propose the "distributed dining philosophers problem." This problem involves families of philosophers that borrow forks from their neighbors, where different families have different responses when forks are not immediately available.

[**Wirth 77**] Wirth, N., "Toward a Discipline of Real-Time Programming," *CACM*, vol. 20, no. 8 (August 1977), pp. 577–583. This paper discusses the problems of real-time and concurrent programming. Wirth argues that real-time programs should first be designed as time-independent systems and then modified to satisfy temporal requirements.

Wirth introduces the language Modula for describing real-time systems. Modula resembles Concurrent Pascal in both design and intent. In addition to constructs that parallel the classes, processes, and monitors of Concurrent Pascal, Modula has a type of object for performing input and output. Whereas Concurrent Pascal prohibits simultaneous access to a shared variable, Modula does not. Like Edison [Brinch Hansen 81], Modula is a language that can be used to ensure security but does not demand it. Modula also leaves many scheduling decisions to the programmer.

[**Wirth 82**] Wirth, N., *Programming in Modula-2*, Springer-Verlag, New York (1982). Modula is a complex language. Wirth has designed a simpler successor, Modula-2. This book is the reference manual for Modula-2.